Shortcomings with Using Edge Encodings to Represent Graph Structures

Gregory S. Hornby

University of California Santa Cruz
Mail Stop 269-3, NASA Ames Research Center
Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov
http://ti.arc.nasa.gov/people/hornby/

Abstract

There are various representations for encoding graph structures, such as artificial neural networks (ANNs) and circuits, each with its own strengths and weaknesses. Here we analyze edge encodings and show that they produce graphs with a node creation order connectivity bias (NCOCB). Additionally, depending on how input/output (I/O) nodes are handled, it can be difficult to generate ANNs with the correct number of I/O nodes. We compare two edge encoding languages, one which explicitly creates I/O nodes and one which connects to pre-existing I/O nodes with parameterized connection operators. Results from experiments show that these parameterized operators greatly improve the probability of creating and maintaining networks with the correct number of I/O nodes, remove the connectivity bias with I/O nodes and produce better ANNs. These results suggest that evolution with a representation which does not have the NCOCB will produce better performing ANNs. Finally we close with a discussion on which directions hold the most promise for future work in developing better representations for graph structures.

1 Introduction

In the field of evolutionary computation, many problems are cast as the evolution of graph structures. The two most common of these are the evolution of artificial neural networks (ANNs), such as for controllers for robots [28], and the evolution of circuits [24,35]. As yet most work has achieved only simple circuits with few components and ANNs capable of only basic behaviors. For this work to be of practical use it is necessary to improve the evolutionary ability of these systems to scale to more complex and interesting results.

Preprint submitted to Genetic Programming and Evolvable Machines31 May 2006

To evolve more complex and powerful ANNs and circuits the evolvability and scalability of the representation used to encode them become increasingly more important. For a number of years it has been recognized that representing designs with a genotype that directly encodes the solution will not scale to complex structures because of the exponential growth in the size of the design space [1,4,6,15,17]. The alternative to evolving with a direct encoding is to use a representation that indirectly specifies it, such as by using a genotype which consists of program for building the desired type of graph structure.

Of the different indirect representations that have been developed for encoding graph structures one that has shown much promise is edge-encoding (EE) languages, which encodes a graph with a program of graph-construction operators that act on the edges of a graph [25]. In this paper we point out three different shortcomings of edge-encoding languages. The main problem we identify with EE is the node creation-order connectivity bias (NCOCB). EE has this bias because the connectivity of a node is determined by the number of graph construction operators that are beneath it in the tree-structured genotype: at one extreme half the nodes in a typically created graph will have a single input and a single output since they were produced from leaf operators in the genotype and at the other extreme one node will have roughly twice the number of connections as the next most highly connected node. Another problem with EE occurs if special node-construction operators are used for creating input/output (I/O) nodes, in which case most randomly created individuals will not have the desired number of I/O nodes and as evolution proceeds the variation operators will have a high probability of changing the genotype so that it produces networks with incorrect numbers of I/O nodes.

One way to address the problems of producing the correct number of I/O nodes and the NCOCB with I/O nodes is by changing the construction language. Rather than having operators in the language for creating a new I/O node, or assigning the nth created node as the ith I/O node, an alternative is to start with the desired number of I/O nodes and then use parameterized-connection operators for adding edges to these nodes. Problems in creating and maintaining networks with the correct number of I/O nodes are reduced since all networks start with the desired number and no operators exist for creating/removing them. Also, with parameterized connection operators the expected number of connections for all I/O nodes is equal for randomly created genotypes and does not suffer from the NCOCB.

The rest of the paper is organized as follows. First, a review of the shortcomings of other representations for graph structures is given so as to motivate our focus on edge encodings. This is followed by a description of a canonical method for using edge encoding to represent artificial neural networks as well as two methods for connecting to I/O nodes. Next we present our experiments which show the different biases with edge-encoding operators and

demonstrate that evolution with parameterized operators for connecting to I/O nodes is better at producing networks that solve 3-parity and produces better controllers on a computer-simulated goal-scoring task. Since the parameterized edge encoding does not have the NCOCB on I/O nodes this suggests that evolution with a representation that does not have the NCOCB at all may perform even better. Finally we close with a discussion on the underlying problem with constructing graphs from tree-structured representations using edge operators and conclude with a summary.

2 A Review of Representations for Graph Structures

To understand why EE has become a popular choice for evolving graph structures it is worth reviewing the other representations for encoding graph structures and the shortcomings that these representations have.

As an aid to understanding the shortcomings of other representations they are categorized by using a classification scheme for predicting the scalability of a representation based on its properties of combination, control flow and abstraction [13,15,17], which is summarized in table 1. To indicate the combinative properties of a representation, the first column of the table specifies the structure in which building blocks are assembled (eg. parameters, strings, trees, matrices). For these entries plural vs. singular is used to distinguish between a representation which has a single string/tree/matrix (eg. DCGP) or those which maintain a collection of strings/trees/matrices (eg. GENRE). The remaining columns are for the properties of Control Flow (Iteration and Conditionals) and Abstraction (Labeled procedures that can be called, Parameters to these procedures, and the ability to call these labeled procedures Recursively). Using these properties *open-ended* representations, those which are not merely parameterizations, can be placed into three broad categories. Depending on whether or not a representation has reuse (such as through iteration or abstraction) the class of open-ended representations is divided into *generative* representations, which allow for reuse of the genotype in creating the phenotype, and *non-generative* representations. Non-generative representations can also be further divided into *direct* representations in which the genotype being evolved is the phenotype and *indirect* representations in which there is some processing performed on the genotype to produce the phenotype.

There have been many non-generative representations which encode a graph structure either directly or indirectly. Examples of direct representations for encoding graph structures are GNARL [2] and Parallel Distributed Genetic Programming (PDGP) [31]. With GNARL, the genotype is a neural network that is evolved through mutation only and with PDGP a graph-structured computer program is represented by itself. One example of an indirect repre-

sentation is the simple developmental model of Nolfi and Parisi [29,30] in which networks are created by placing neurons on a two-dimensional plane with links between neurons formed through an artificial growth process. Their representation consists of a sequence of gene parameters which include whether or not the neuron exists, its location, and how links branch out from it. Difficulties with this representation are the computational overhead of calculating neuron-link intersections during the construction process and the limited means by which the growth of links is specified. Since none of these systems can hierarchically encapsulate and construct modules or reuse elements of the genotype none of these systems will scale to large networks.

Possibly the first example of a generative representation for encoding ANNs is Kitano's L-system for matrices [19]. In Kitano's system each production rule rewrites a symbol with a 2 by 2 matrix, unlike traditional L-systems which operate on strings of symbols [23]. While this matrix L-system seemed to perform well in comparison with a direct representation, a later investigation found it to be no better [33]. Possible reasons why this system does not outperform a direct representation is the limitations of the rules: each rule specifies the replacement of a symbol with a 2x2 matrix, which is a parameterized construct rather than an open-ended construct that allows for the combination of primitives to create more complex rules; and the rules are not able to call each other recursively which limits their reusability. An additional shortcoming of this approach is that it is difficult to ensure that the final weight matrix has continuous rows and columns that go all the way across the matrix.

Recursion is allowable in two other representations based on L-systems by using strings instead of matrices. With the system of Boers et al. [5] each alphabetic symbol in the L-system represents a neuron in the ANN and a grouping of symbols inside brackets are used to specify a sub-network. Connections are made between adjacent neurons, or modules, and these are always forward connections. Shortcomings with this approach are that it only encodes feed forward networks and does not have a method for specifying edge weights or the properties of a neuron. In the work of Haddow et al. they evolve circuits by using context sensitive L-systems as a generative representation for binary strings which specify the LUT logic tables of Field-Programmable Gate Arrays (FPGAs) [12]. While this representation has reuse, and thus is a generative representation, it is not clear that reusing the same set of bits in different parts of a LUT configuration table is a good form of reuse since a reused block of bits will have a different phenotypic effect if it is placed in a different part of a LUT configuration table.

With Miller and Thomson's Developmental Cartesian Genetic Programming (DCGP) an integer based genotype is used to encode a graph structure [27]. The integers in the genotype refer to nodes in the graph and after all nodes have been created the connections are made using the modulo operator (with

the total number of nodes in the graph as the base) to determine which node an integer refers to. While this type of encoding does not have the NCOCB, a problem with it is that if the number of nodes change, through either mutation or recombination, then all the references to nodes will change since the base for the modulo operator will be a different number. Consequently it is difficult to alter the network topology without also radically changing the network's behavior. Not surprisingly they found that their developmental system for evolving circuits performed worse than a direct representation on a binary adder problem and concluded that their representation is more difficult to evolve than a direct representation.

The first tree-structured representation for encoding graphs is Gruau's cellular encoding (CE) [11]. With this representation each node of the tree is a graph construction operator that changes the graph by performing an operation on one of the nodes. Iteration is present through a recurse node for re-executing the tree, and abstraction is implemented through labeled subtrees, which are called automatically defined sub-networks (ADSNs) [11]. Advantages of CE's indirect, tree-structured representation are that it better allows for variable sized graphs than directly using a weight matrix, and GP style recombination between two trees is easier and more meaningful than trying to swap sub-networks with a graph-structured representation. Also, the representation is generative with both iteration and abstraction implemented. These advantages led to a number of variations on CE by using different graph construction operators [8,9].

Yet Luke and Spector noted that CE has serious drawbacks because its graph-construction operators act primarily on nodes [25]. One shortcoming is that operators are execution-order dependent and, as a result, swapping subtrees in the genotype does not result in a swapping of subgraphs of the phenotype. A second shortcoming is that operators on nodes can create an arbitrary number of edges – splitting a node in two will create a second node with the same inputs and outputs – which is problematic for specifying the weights of all the newly created edges. The third shortcoming of CE is that it has a strong bias toward producing very dense graphs, which makes it ill-suited for evolving circuits.

As an alternative to CE, Luke and Spector proposed a graph construction language called edge encoding (EE) in which operators act on edges instead of on nodes [25]. Advantages of EE are that at most one link is created with a construction operator, which allows either the leaf nodes of the genotype or the construction operator itself to specify the weight to attach to that link, and, unlike with CE, sub-trees of construction operators will create the same sub-network independent of where in the construction-tree they are located. In their implementation they used labeled sub-trees and implemented recursion without parameters by setting a maximum recursion depth. Previously we used EE as the language for encoding ANNs in GENRE and used it to evolve ANNs

Table 1
Properties of different open-ended representations for encoding graphs.

	Structure	Control Flow		Abstraction		
System		Iter.	Cond.	Labels	Param.	Recur.
Direct						
AS&P (GNARL) [2]	graph	no	no	no	no	no
Poli (PDGP) [31]	graph	no	no	no	no	no
Indirect						
N&P [30]	parameters	no	yes	no	no	no
Generative						
BKH&S [5]	strings	no	no	yes	no	yes
Gruau (CE) [10]	trees	yes	no	yes	no	no
HT&vR [12]	strings	no	yes	yes	no	no
Hornby (GENRE) [13]	strings	yes	yes	yes	yes	yes
Kitano [19]	parameters	no	no	yes	no	no
L&S (EE) [25]	trees	no	no	yes	no	yes
M&T (DCGP) [27]	string	yes	yes	no	no	no

to calculate parity and as controllers for robots [13,17]. This form of graph construction language has become common in representations for constructing circuits and is frequently called CE instead of EE [21].

Of these different generative representations for encoding graph structures, only edge encoding has yet to be shown to have serious drawbacks. The focus of the rest of this paper is an analysis of edge encoding languages and its shortcomings for representing graph structures.

3 Edge Encodings

To demonstrate the shortcomings of EE, in the following sections experiments will be performed using them to encode ANNs. The ANNs that are used are recurrent networks similar to those of Beer and Gallagher [3], and of our previous work [16,18]. Each non-input neuron has an input bias, θ, and a time constant, τ. The activation value of a non-input neuron a_i at time t is:

$$a_{i,t} = \tau_i a_{i,t-1} + (1-\tau) tanh\Big(\sum_j W_{ji} a_{j,t-1} + \theta_i\Big) \quad (1)$$

The activation values of input neurons are given as inputs to the network.

The different methods for representing graphs with EE all start with an initial graph configuration and then new nodes and edges are added by executing the operators in the assembly procedure. The following list is a typical set of graph-construction operators in an edge-encoding language, for which the operators are acting on the edge connecting from A to B:

- `add_reverse`: creates a link from B to A.
- `add_split`(n): creates a new neuron, C, adds a new link from A to C and creates a new edge connecting from neuron C to neuron B. The bias of neuron C is set to $\theta = n$, and its time constant is set to zero.
- `add_split_cont`(m, n): acts the same as `add_split()`, only it creates a neuron with a bias of $\theta = m$ and a decay constant of $\tau = n$.
- `connect`: creates a new link from A to B.
- `dest_to_next`: changes the to-neuron in the current link to its next sibling.
- `loop`: creates a new link from neuron B to itself. The `no-op` operator does nothing.
- `set_weight`(n): sets the weight of the current link to n.
- `source_to_next`: changes the from-neuron in the current link to its next sibling.
- `source_to_parent`: changes the from-neuron in the current link to the input-neuron of the current from-neuron.

Of these operators `add_split`(n) and `add_split_cont`(m, n) have exactly three children operators since after their execution the edge they act on becomes three edges. The `set_weight`(n) operator has no children, consequently it is always a leaf node and the `no-op` has either zero or one children so it can be either a leaf node and halt the development of the graph on the current edge, or it can be used to delay execution on the current edge for a round allowing time for the rest of the graph to develop more. Execution of the rest of the operators results in the addition of a single new edge to the graph so they have exactly two children operators: one to continue graph construction along the original edge and one operator to continue construction along the new edge.

Using the operators described above the sequence of graphs from figure 1.b-1.i illustrates the construction of a network from the genotype in figure 1.a. Graphs are constructed from this genotype by starting with a single neuron linked to itself, figure 1.b, and executing the operators in the assembly procedure in breadth-first order. First, executing `split(0.01)` adds node `b` with a bias of 0.01 and pair of links, figure 1.c. The operator `set-weight(0.1)` sets the weight of the link $\overrightarrow{aa}$ to 0.1, `no-op` performs no operation, and then `split(0.02)` results in the creation of neuron `c`, with a bias of 0.02, and two more links, figure 1.d. `Source-to-parent` creates a second link, $\overrightarrow{ab}$, and `set-weight(0.2)` sets the weight of the link $\overrightarrow{ba}$ to 0.2, figure 1.e. The second

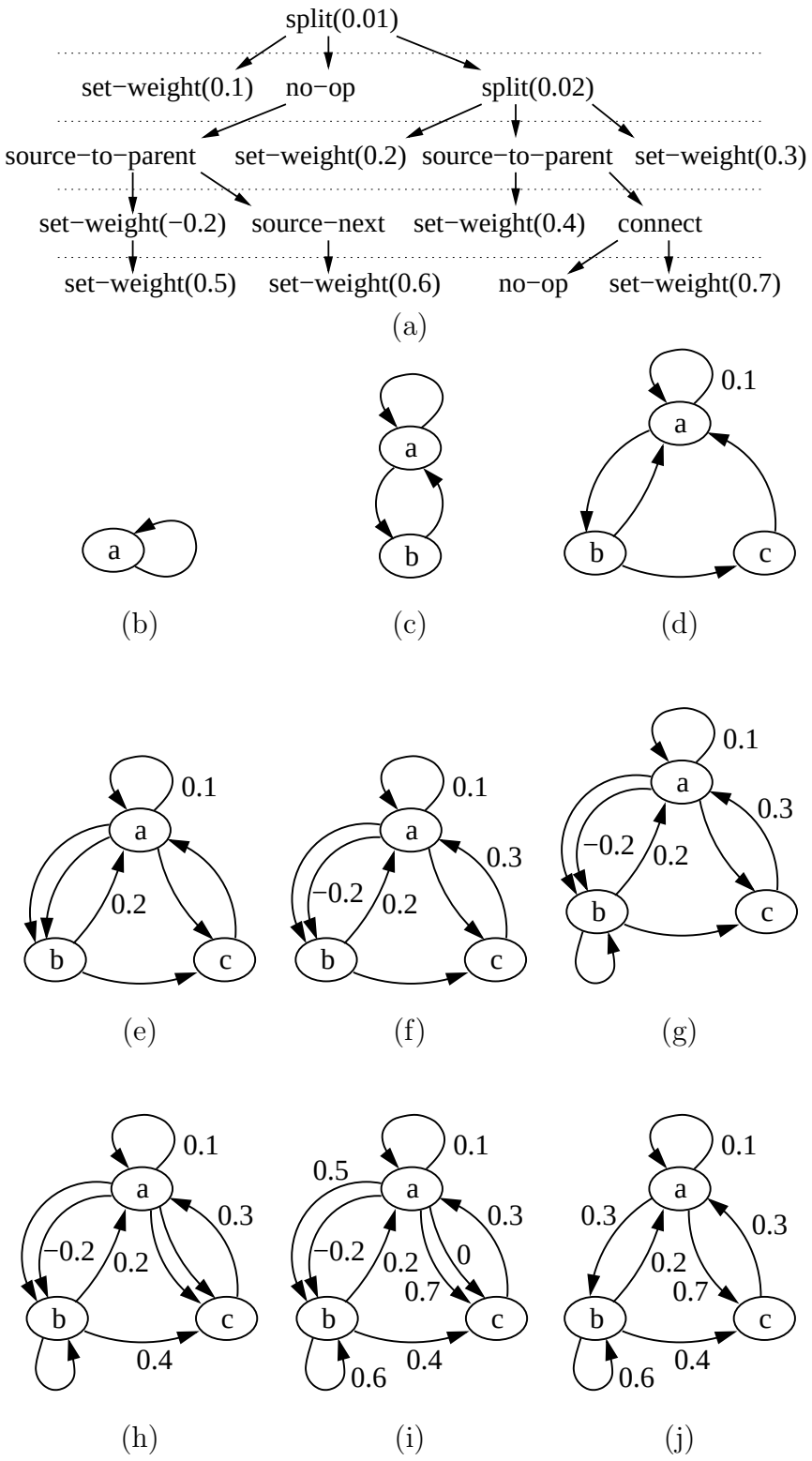


Fig. 1. A tree-structured encoding of a network (a), with dashed-lines to separate the layers, and (b-j) construction of the network it encodes.

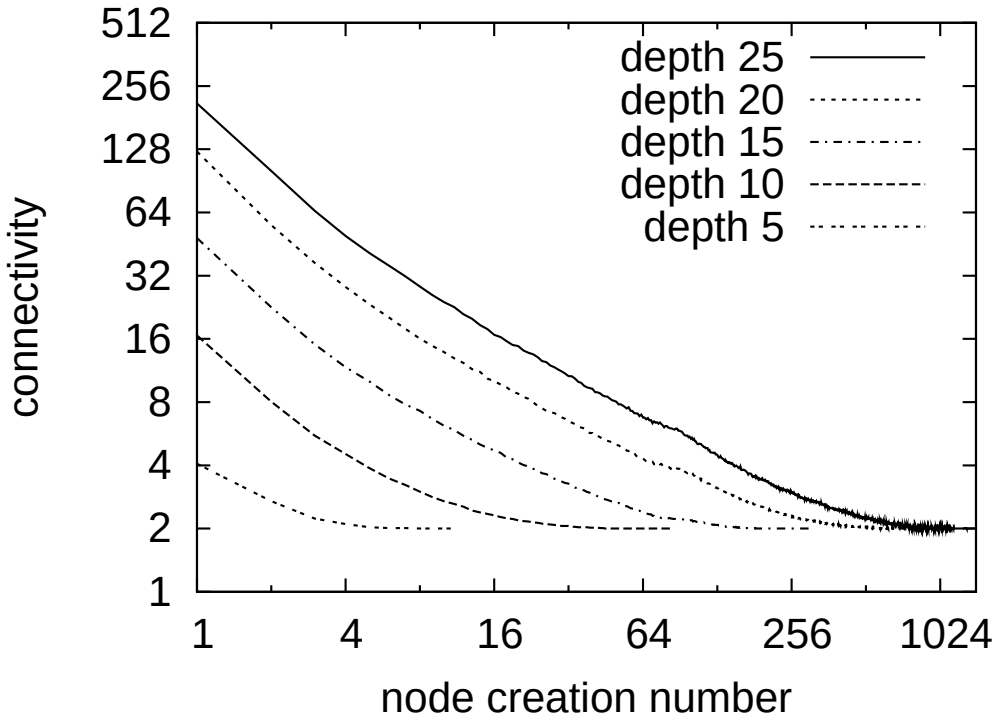
`source-to-parent` operator creates the link $\overrightarrow{ac}$, executing `set-weight(0.3)` sets the weight of the link $\overrightarrow{ca}$ to 0.3 and `set-weight(-0.2)` results in a weight of -0.2 assigned to the link $\overrightarrow{ab}$, figure 1.f. The `source-to-next` operator results in the link $\overrightarrow{bb}$ being created, figure 1.g. The operator `set-weight(0.4)` sets the weight of link $\overrightarrow{bc}$ to 0.4 and then executing `connect` creates an additional link $\overrightarrow{ac}$, figure 1.h. Executing `set-weight(0.5)` sets the weight of link $\overrightarrow{ab}$ to 0.5, `set-weight(0.6)` sets the weight of link $\overrightarrow{bb}$ to 0.6, `no-op` sets the weight of link $\overrightarrow{ab}$ to 0.0, and `set-weight(0.7)` sets the weight of link $\overrightarrow{ac}$ to 0.7, figure 1.i. After all tree-construction operators have been executed, there is a post-processing phase that consolidates the weights of links with the same source and destination nodes, figure 1.j. In addition, when evolving artificial neural networks with I/O nodes, there is an additional post-processing pruning step that removes from the ANN all neurons that are not on a directed path to an output neuron. Both of these post-processing phases are used to simplify the ANN so as to reduce the amount of computation needed to update it.

4 Node Creation Order Connectivity Bias

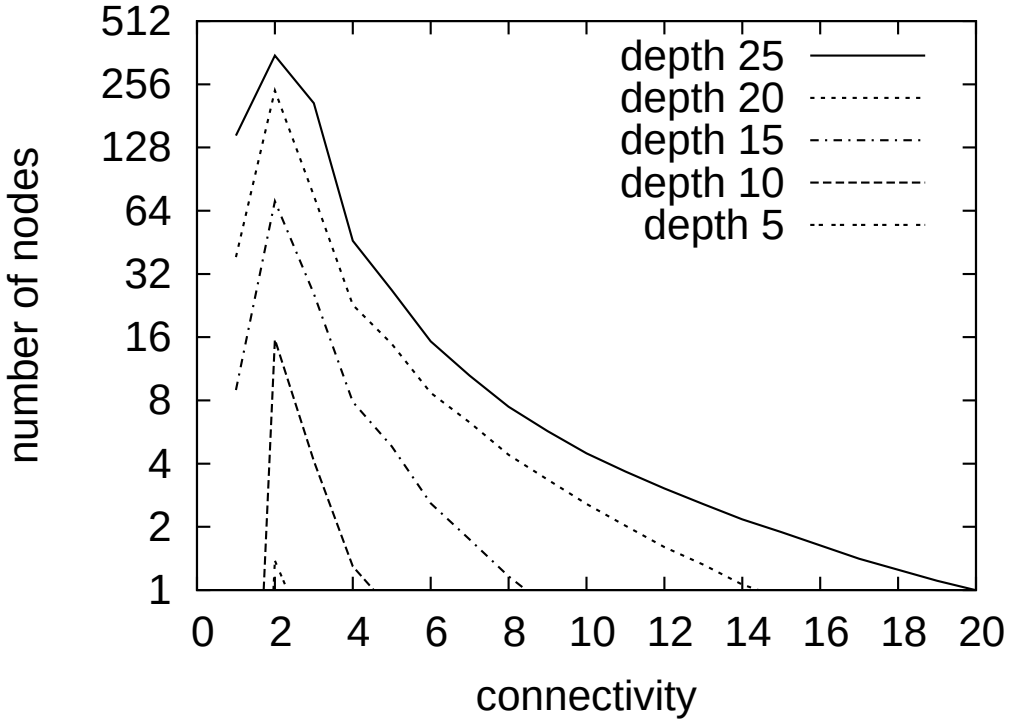
A problem with using EE operators and tree-structured assembly procedures is that nodes created from operators early in the construction process tend to have a greater connectivity (sum of number of edges into and out of them) than nodes created later in the process. We call this bias the node creation-order connectivity bias (NCOCB) and it can be demonstrated by examining the average connectivity of randomly created, edge-encoded graphs.

The graph in figure 2.a contains plots of the average connectivity of graphs created with graph-constructing, tree-structured genotypes of different depths. The lines in the graph are plots of the average connectivity for the ith created node averaged over ten thousand randomly created genotypes for trees of depths five, ten, fifteen, twenty and twenty-five. From this graph it can be seen that there is an inverse-exponential relation between node creation order and its connectivity: the first node that is created has a higher connectivity than every other node in the graph and the last half of the nodes created having a connectivity of two.

Because the connectivity of a node is strongly based on its height and since there is a bias in the distribution of the heights of operators in a tree-structured genotype there is a bias in the distribution of node connectivities, figure 2.b. There will be one node in the graph with a connectivity higher than all the others since there is only one operator at the top of the tree-structured genotype and roughly half the nodes in the graph will have a connectivity of two (one input and one output edge) since around half the operators in the geno-



(a)



(b)

Fig. 2. Graphs of (a) the average node connectivity by order of creation, and (b) the number of nodes with a given connectivity for randomly created individuals of different tree depths.

type are leaf nodes and the nodes they create in the graph have a connectivity of two.

One consequence of this bias is that I/O nodes that are created early in the construction process will have a significantly higher number of output/input edges than those I/O nodes created at the end of the construction process. Thus if I/O nodes are created by the tree-structured assembly procedure, the first I/O nodes will have significantly more inputs/outputs from/to them than those created later in the construction process. For input nodes, this suggests that the first inputs are likely to have a greater influence on the behavior of an ANN than the latter inputs and for output nodes this suggests that there is more processing of inputs in calculating the activation values of the first output nodes than for the latter output nodes. In the following section we will discuss two methods for handling I/O nodes, one of which does not have the NCOCB.

5 Handling Input/Output Nodes

A consideration with adapting EE to representing ANNs is determining how to specify the input and output (I/O) nodes. One approach is to assign the first n nodes to be the inputs and the next m nodes to be the outputs. Yet with this approach, if the genotype is modified such that a node is added/deleted, then the function of all I/O nodes after that will shift and it is unlikely that the resulting ANN will perform well. A better method for handling I/O nodes is to add operators to the language for creating I/O nodes, and we call this method a constructive edge-encoding language (CEEL). [1] Since this method has problems in creating the correct number of I/O nodes and also has a node creation-order connectivity bias (NCOCB) another approach is to use a parameterized edge-encoding language (PEEL). With PEEL, graph construction starts with a single hidden node and edge along with the desired number of I/O nodes. The hidden nodes and edges in the graph are constructed as with CEEL, but connections to the I/O nodes are made through parameterized I/O connection operators. We now describe CEEL and PEEL in greater detail.

Constructive edge encoding language (CEEL) extends the EE language of the previous section by using specialized operators for creating I/O nodes. There is one operator for creating input nodes and one operator for creating output nodes:

- `add_input`: creates a new neuron, C, which is an input neuron, and adds a

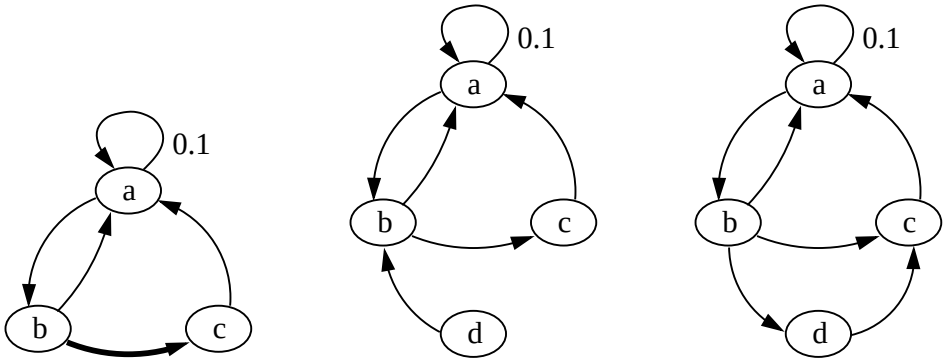
[1] Previously this language was called SEEL, for Standard Edge Encoding Language[14].

new link from C to A.

- `add_output_split`(n): performs a `split`(n) operation with the newly created neuron set to an output neuron. The bias of this new node is set to $\theta = n$ and the time constant is set to $\tau = 0$.

In the above operator descriptions the edge that is being acted on connects from node `A` to node `B`. Since `add_input` adds one new edge to the graph this operator has two successors, one successor to continue construction on the original edge and one successor to start construction on the new edge. As with the `split`(n) operator, the `output_split`(n) operator has three successors to continue graph construction along each of the three resulting edges.

Examples of the two CEEL I/O construction operators are shown in figure 3. Figure 3.a consists of a graph with the current edge, $\overrightarrow{bc}$, in bold. If the next operator to be executed is `add_input`, the result is the graph shown in figure 3.b. Alternatively, if the next operator is `add_output_split`(0.25) the resulting graph is that shown in figure 3.c.



(a) (b) (c)

Fig. 3. Adding an I/O node: (a) the original graph with the active edge connecting from `b` to `c`; (b) the resulting graph after executing `add_input`; (c) the resulting graph if `add_output_`(0.25) is executed instead.

As was discussed in section 4, EE suffers from a connectivity bias based on the order in which nodes are created and since CEEL creates I/O nodes as part of the construction process this connectivity bias affects I/O nodes that are created during the construction process. A method to remove the connectivity bias with I/O nodes is to have these nodes exist in the initial graph and then adding connections to them with parameterized connection operators:

- `connect_input`(i): creates a link from the ith input neuron to B.
- `connect_output`(i): creates a link from B to the ith output neuron.

Since each of these operators creates a new edge, both operators have exactly two children operators: one to continue network construction along the original edge and one to construct along the new edge. We call this language PEEL,

for Parameterized Edge Encoding Language. With PEEL, the connectivity bias of the I/O nodes is dependent on the bias in generating the parameters to the I/O connection operators and there will be no connectivity bias for I/O nodes if there is a uniform distribution in generating parameter values.

Given that PEEL does not have the NCOCB with I/O nodes and, as we will show in the following sections, is more robust to creating and maintaining ANNs with the desired number of I/O nodes it is worth noting that there are situations in which CEEL would be the preferred method for handling I/O nodes. One example of this is when the number of I/O nodes is not known ahead of time, such as when an individual encodes for the physical structure of a robot, with a variable number of actuators and sensors, along with the ANN controller [17].

6 Comparing CEEL vs. PEEL on Producing Valid ANNs

To demonstrate that PEEL is better than CEEL for creating and maintaining genotypes that produce ANNs with the correct number of I/O nodes we present experiments comparing the two languages. In these experiments we compare the number of valid ANNs that are created using the two edge encoding languages. A network is considered valid if it has four input neurons and four output neurons (arbitrary values selected for this experiment) and for each input neuron there is a path to at least one output neuron and each output neuron is on a path from at least one input neuron.

Table 2 shows the probability that a randomly created individual with a genotype of a given depth will construct a valid network, with these values based on generating ten thousand random individuals for each configuration. With CEEL the probability of creating a valid network had a maximum value of just under 2% for trees of depth seven whereas with PEEL the probability of creating a valid network asymptotically increases beyond 88% as the tree depth increases beyond 13. Thus valid networks are more likely to be created with PEEL than with CEEL. The reason PEEL does not score 100% even though it starts with the correct number of I/O neurons is because some input nodes may not be on a path to an output node and are pruned in the post-processing phase of ANN construction.

In addition to the problem of creating initial individuals with the correct number of I/O nodes, CEELs have difficulty maintaining these numbers under mutation and recombination. To show that PEELs better maintain valid networks we compare the number of networks that still have four inputs and four outputs after mutation and recombination from valid parents. For this comparison the mutation operator modifies an individual by changing one symbol

Table 2
Percentage of valid networks generated out of ten thousand randomly created tree-structured assembly procedures.

Depth	≤4	5	6	7	8	9	10	11	12	13
CEEL	0	0.03	1.03	1.83	0.93	0.34	0.13	0.06	0.02	0
PEEL	0	0	0.12	3.14	19.7	46.6	67.3	80.7	86.4	88.5

with another, perturbing the parameter value of a symbol, adding/deleting some symbols, or recombining an individual with itself. Two types of recombination are used, with equal probability of using one or the other. The first recombination operator is the standard GP recombination that swaps random subtrees between parents [20]. The second recombination operator is similar to one-point crossover [32] and we call it matched recombination (MR). MR works by lining up two trees and, starting at the root, matches up the children nodes by type and argument values, finds the locations at which subtrees differ and then picks one of these places at random to swap, similar to homologous crossover [7,22].

Since random trees of depth seven produced the most valid networks with CEEL, we compared ten thousand mutations and recombinations between CEEL and PEEL on valid, randomly created individuals. With CEEL, 15.2% of mutations and 20.8% of recombinations resulted in the offspring having an incorrect number of I/O nodes. In comparison, with PEEL the failure rate is 6.5% with mutation and 10.5% with recombination. These results show that networks encoded with PEEL are twice as robust to variation operators than those encoded with CEEL.

7 Comparing CEEL vs. PEEL on Calculating 3-Parity

Having shown that PEEL is better suited for creating and maintaining valid networks that are generated at random, we are interested in determining if this continues to be the case on individuals that are being evolved and also if there is a difference in evolutionary performance. To see if this is the case, we evolve ANNs to solve the odd-3-parity function using CEEL and PEEL to encode these networks. The odd-3-parity function returns `true` if the number of `true` inputs is odd and returns `false` otherwise. This function is difficult because the correct output changes for every change of an input value. In addition, the even/odd-n-parity functions have become a standard benchmark function in genetic programming (GP) and past experiments have shown that GP does not solve the five-parity (or higher) problem without automatically defined functions [20].

For these experiments a population of 200 individuals was evolved for 500 generations using a generational evolutionary algorithm. To create the initial population individuals were generated randomly until each individual in the population encoded a network with exactly three inputs and one output. Parents were selected for reproduction using remainder-stochastic sampling based on rank, using exponential scaling with a scaling factor of 0.035. New individuals were created by randomly deciding to use either mutation or recombination with equal probability of picking either.

An individual is evaluated by testing the ANN it encodes on each of the 2^3 possible inputs, using an input value of 1.0 for `true` and -1.0 for `false`. Networks are updated up to four times with the value of the output neuron examined after each update iteration to determine the parity value calculated for that set of input values. If the value of the output neuron is greater than 0.9 then the output of the network is taken as `true` and if the value of the output neuron is less than -0.9 then the output of the network is taken as `false`. If the value of the output neuron is between -0.9 and 0.9 then the network is updated again until its output value is either greater than 0.9 or less than -0.9, for a maximum of three additional updates. The network receives a score of 2.0 for returning the correct parity value and a score of -1 for an incorrect answer. If, after four network updates, the value of the output neuron is between -0.9 and 0.9 then there is no further updating of the network and the individual receives a score of 1.0 for this test if either the value of the output neuron is positive and the parity was `true` or if the value of the output neuron is negative and the parity is `false`. No penalty is given for having an incorrect value in this case. The fitness value of an individual is the sum of its scores on all eight possible inputs with the maximum fitness being 16.

Out of 500 evolutionary runs with each CEEL and PEEL, evolution with CEEL solves the 3-parity problem 127 times and evolution with PEEL solves it 327 times. Evolved networks were recurrent with sizes in the range of 5 to 40 nodes. The evolutionary performance with both encodings is shown in figure 4, which contains plots of the best individual in the population averaged over the 500 evolutionary runs. The final average values are (mean$\pm$s.e.): CEEL, 12.45$\pm$2.46; PEEL, 14.25$\pm$2.55. Using a two-tailed Mann-Whitney test the difference in performance is highly significant with $P < 0.001$.

In the previous section it was shown that PEEL was better at producing valid networks than was CEEL, we now examine whether or not this holds true over the course of evolution. The graph in figure 5 plots the percentage of offspring created with mutation and recombination that still encoded ANNs with 3 inputs and 1 output. These graphs show that within a couple of generations of evolution the success rate of producing valid ANNs improves considerably over the success rate with applying variation to randomly created networks,

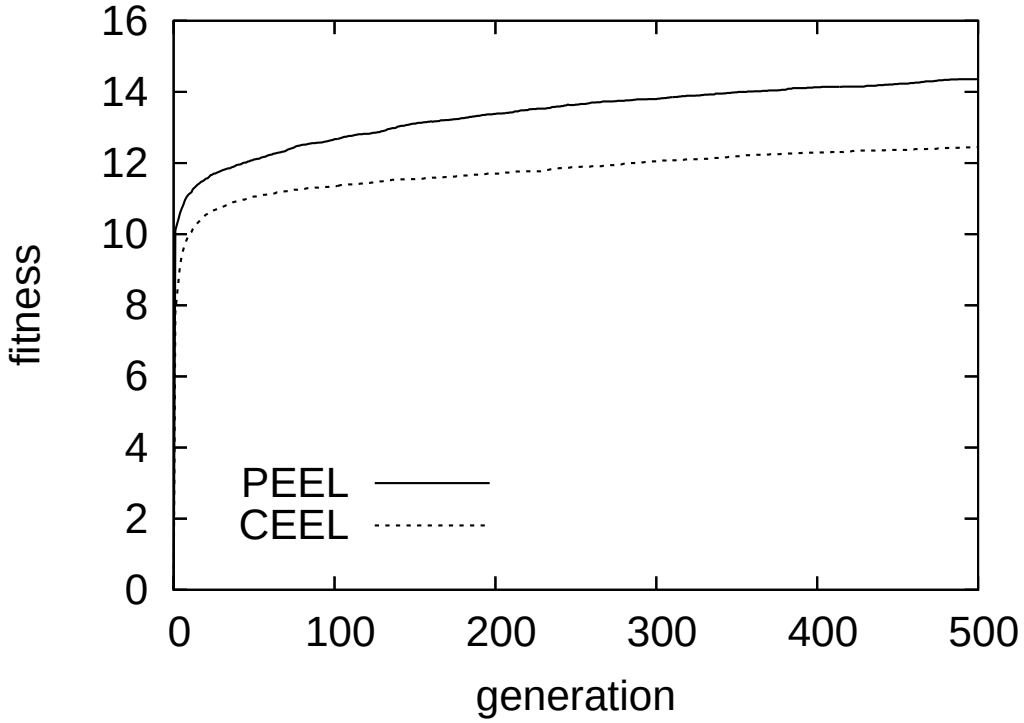


Fig. 4. Fitness of the best evolved ANN averaged over 500 trials.

Table 3
Average connectivity of I/O neurons at different stages of evolution.

	CEEL				PEEL			
Generation	I/O 1	I/O 2	I/O 3	I/O 4	I/O 1	I/O 2	I/O 3	I/O 4
0	2.58	1.68	1.36	1.23	1.48	1.48	1.48	2.20
10	4.29	1.74	1.32	1.14	1.56	1.57	1.56	2.23
50	4.48	1.79	1.33	1.17	1.98	2.04	1.91	2.83
100	4.59	1.82	1.39	1.17	2.24	2.34	2.13	3.18
250	4.75	1.86	1.45	1.22	2.42	2.61	2.29	3.49
500	4.76	1.84	1.49	1.29	2.59	2.78	2.42	3.70

the experiment of the previous section. Yet variation on networks encoded with PEEL is still more than three times as likely to produce valid networks than variation applied to networks encoded using CEEL: in the final generation the success rate with CEEL is 92.9% with mutation and 95.1% with recombination and with PEEL it is 98.0% with mutation and 98.9% with recombination. Since the success rate is both fairly close between CEEL and PEEL and quite high, it would seem that a likely reason for the large and significant performance difference between CEEL and PEEL is the NCOCB.

While the NCOCB was shown to be significant on randomly generated graph

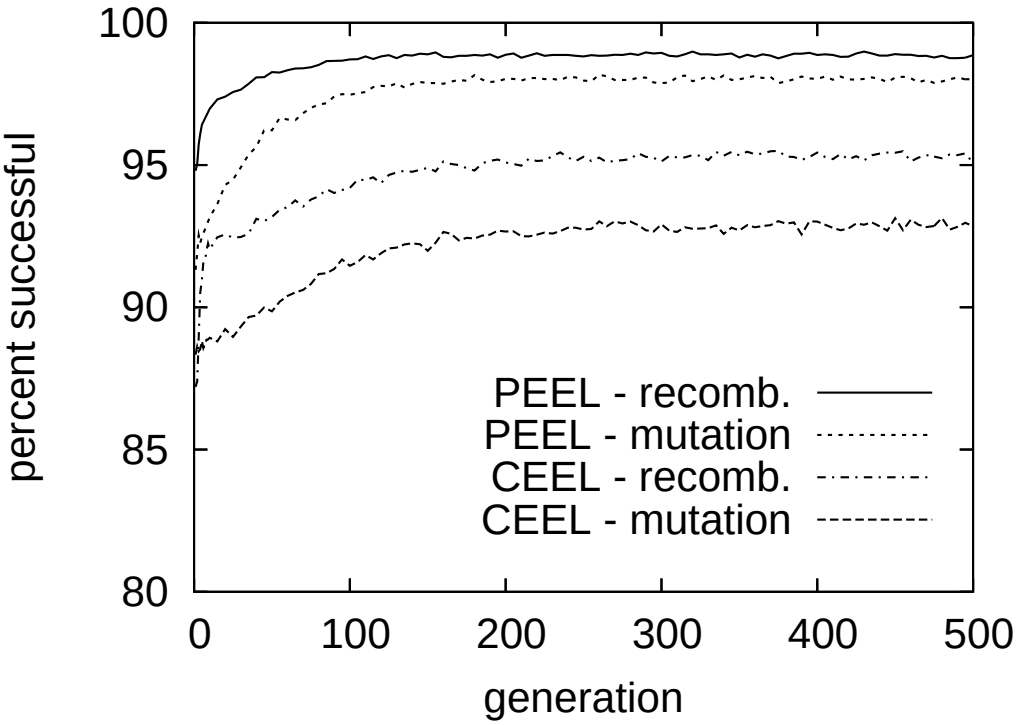
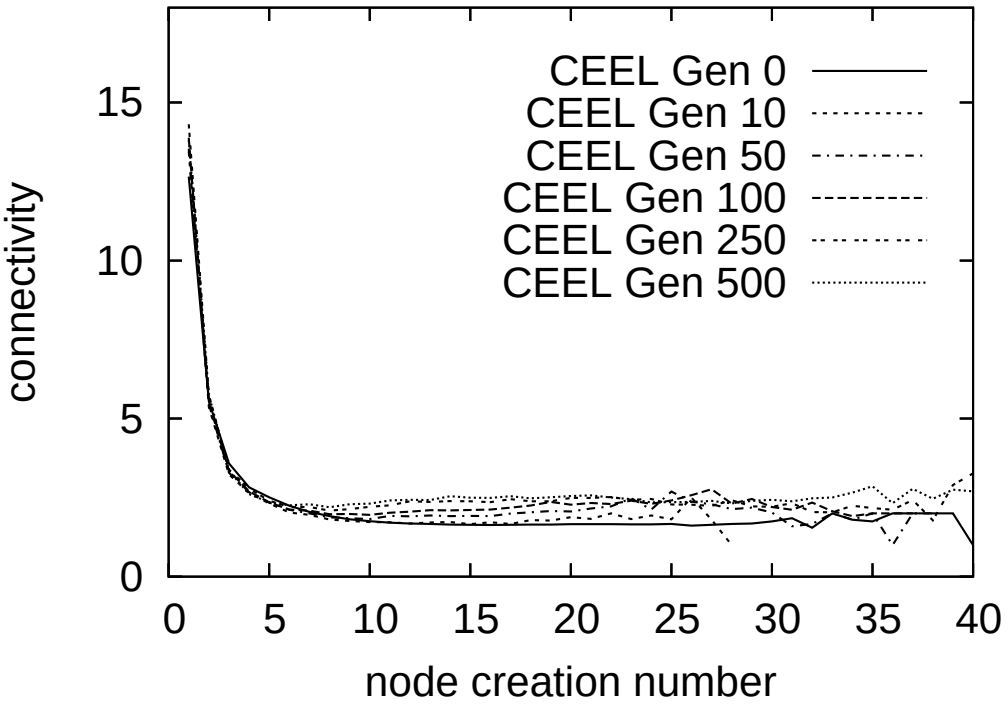


Fig. 5. Percentage of networks that are valid after mutation and recombination from 500 evolutionary runs with a population size of 200 and an equal probability of applying mutation or recombination.

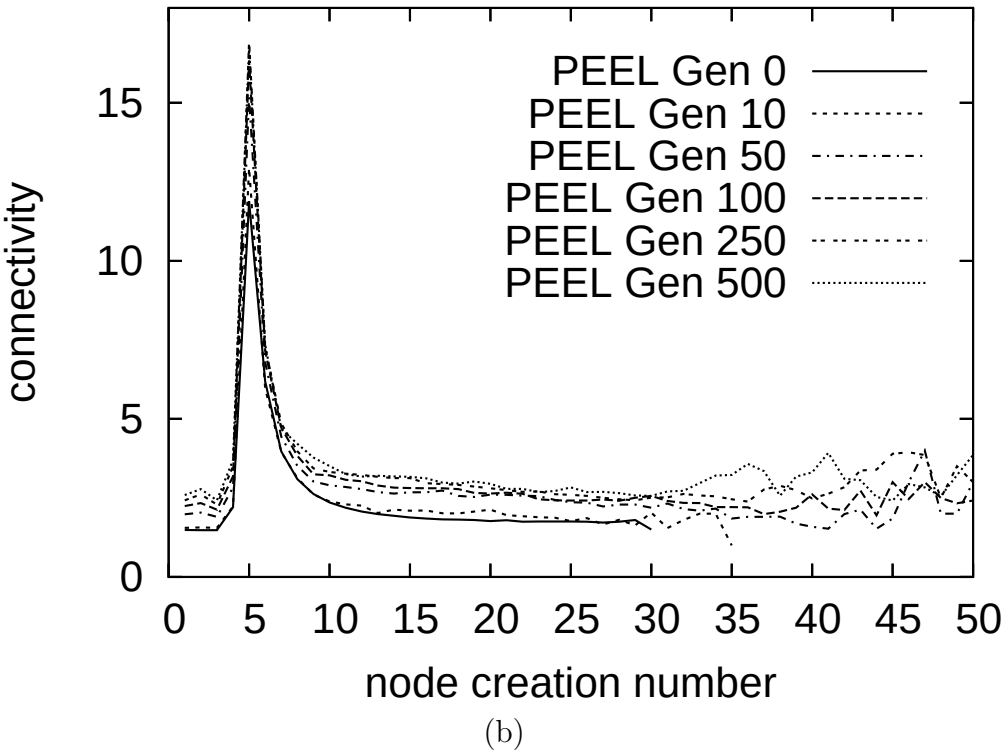
Table 4
Average node location of I/O neurons at different stages of evolution.

	CEEL				PEEL			
Generation	In 1	In 2	In 3	Out 1	In 1	In 2	In 3	Out 1
0	3.98	6.82	9.60	6.53	1	2	3	4
10	4.01	6.57	9.16	3.77	1	2	3	4
50	3.95	6.38	8.86	3.63	1	2	3	4
100	3.93	6.26	8.69	3.60	1	2	3	4
250	3.82	6.08	8.48	3.53	1	2	3	4
500	3.75	5.89	8.22	3.48	1	2	3	4

encodings, it is of interest to know whether or not the NCOCB is reduced, or removed, over the course of evolution. The graphs in figure 6 are plots of the average connectivity for nodes in graphs at different stages of evolution with both CEEL and PEEL. In both graphs the NCOCB is readily apparent, although the vast majority of graphs had less than 30 nodes so the difference in the average connectivity of nodes beyond node 30 node is not very statistically significant. Note that with PEEL, the first four nodes are the I/O nodes and



(a)



(b)

Fig. 6. Average NCOCB with CEEL and PEEL at different stages of evolution on the 3-parity problem.

are present at initialization so the NCOCB bias starts at node 5 with PEEL.

Since the difference between CEEL and PEEL is in how they handle I/O nodes it is worth examining the difference in NCOCB between CEEL and PEEL on I/O nodes. With the parameterized connection operators of PEEL there is no node NCOCB with its I/O nodes, unlike with CEEL where I/O node 1 always has a higher connectivity than I/O node 2, which always has a higher connectivity than I/O node 3 (table 3). In fact, with CEEL, increasing the connectivity of more important I/O nodes (in this case, the output node) is achieved by moving them to being created earlier in the graph construction process. The entries in table 4 show that with CEEL the average node location of I/O nodes moves to earlier in the construction process over the course of evolution, with the output node very quickly becoming the first I/O that is created.

That neither CEEL nor PEEL achieved close to a 100% success rate on the somewhat trivial problem of evolving 3-parity networks further suggests that neither of these variants of EE is well suited to evolving ANNs since other techniques – such as pure genetic programming [20] – perform far better on this problem.

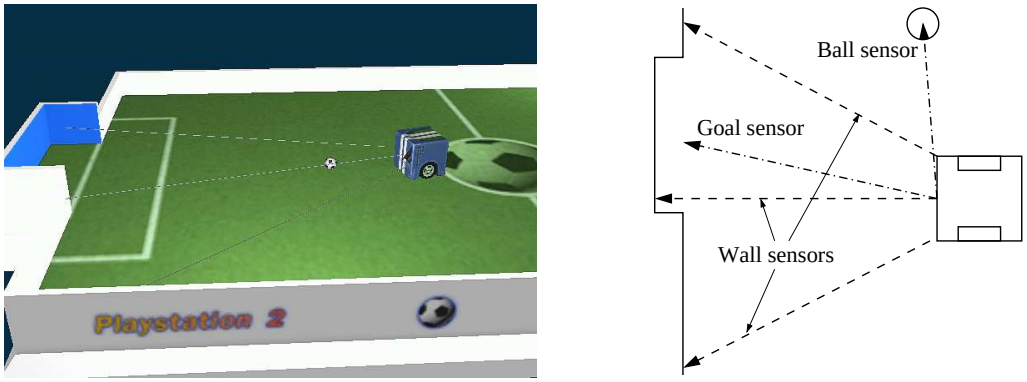
8 Comparing CEEL vs. PEEL on a Goal-Scoring Task

In this last set of experiments, CEEL and PEEL are compared on a control task in which an evolved ANN is evaluated by how well it performs a goal-scoring task. For these experiments the evolutionary algorithm (EA) was run on a cluster of five PlayStation® 2 development systems controlled by a Linux PC. [2] Each experiment consisted of evolving fifty individuals for fifty generations using the same generational EA that was used in the previous set of experiments.

The goal-scoring task consists of driving a two-wheeled soccer-player in a 275x152.5, computer-simulated, walled soccer-field with goals at each end (figure 7.a). This simulator uses a physical dynamics engine based on the model described by Witkin and Baraff [34]. The soccer player has seven sensor inputs and two outputs: input 1 is the angle to the ball, scaled to the range -1 to 1; input 2 is the distance to the ball, scaled to the range 0 to 1; input 3 is the angle to the goal, scaled to the range -1 to 1; input 4 is the distance to the goal, scaled to the range 0 to 1; input 5 is the distance straight ahead, scaled to the range 0 to 1; input 6 is the distance to the left, scaled to the range 0 to 1; input 7 is the distance to the right, scaled to the range 0 to

[2] PlayStation is a registered trademark of Sony Computer Entertainment Inc.

1; output 1 controls the rotational speed of the right wheel, and is scaled to −50 to 50 radians per second; and output 2 controls the rotational speed of the left wheel, and is scaled to −50 to 50 radians per second. For input nodes the scaling of distance values is done by first linearly scaling distances from the range of 0.0 to 250 to values 1.0 to 0.0 (distances greater than 250 are also mapped to 0.0) and then squaring this value to make it more sensitive to nearer distances. An image of the different inputs is shown in figure 7.b.



(a) (b)

Fig. 7. (a) The simulated soccer field and (b) the soccer player and its sensors.

Evaluating an individual consists of eight trials – two each with the ball initially placed in each of the four corners of the field, and the soccer-player placed in the middle of the field – with an individual's overall fitness being the sum of its scores over all eight trials. Initial locations for both the player and ball are perturbed by a small random amount and then the player is given 60 seconds (at 30 frames per second this results in 1800 network updates) to score as many goals as it can. For each goal scored the distance from the vehicle's starting position to the ball plus the distance from the ball's initial location to the goal is added to the individual's score. After a goal is scored the ball is randomly located at the center of the field (with a random perturbation of its location in the range of $x \pm 30$, $y \pm 30$), the minimum distances to the ball and to the goal are reset, and the network is allowed to try to score another goal. Once time expires, an individual's score is increased by how much closer it moved the player to the ball and how much closer it moved the ball to the goal. In addition, if the player scores an own-goal, its score is reduced by the distance it moved the ball from its starting position to its own goal. Evolved individuals encode for recurrent neural networks with anywhere from 10 to 40 neurons and evaluating one generation of fifty individuals took approximately four minutes.

The results of these experiments are plotted in the graph in figure 8 and show that evolution with PEEL produces soccer players that are roughly twice as fit as those evolved with CEEL. The final fitness values of the four trials are: 1239, 1739, 1775, and 1550 with CEEL; and 3615, 2126, 2472, and 3101 with

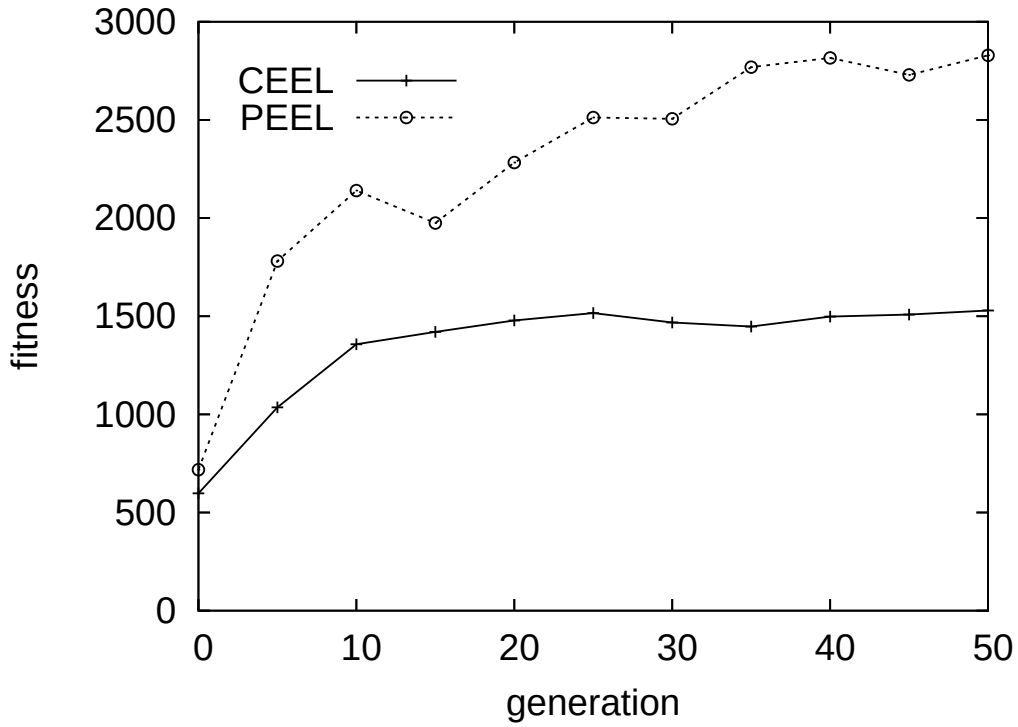


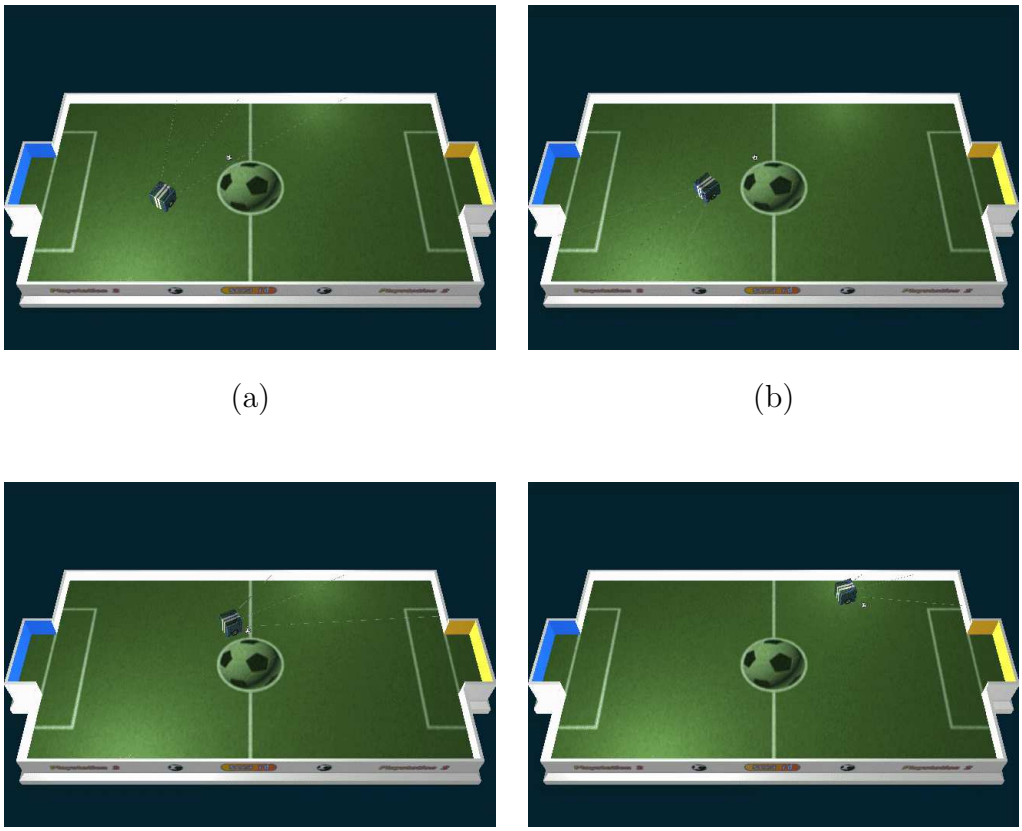
Fig. 8. Fitness of the best evolved goal-scores averaged over four trials.

PEEL. Using a a two-tailed Mann-Whitney test the difference in performance is marginally significant with $P < 0.05$.

The higher fitness of ANNs encoded with PEEL is reflected in the behaviors produced by these networks.[3] ANNs encoded with CEEL produced soccer players that tended to spin in place and move awkwardly or in a looping pattern. These controllers only moved toward the ball somewhat haphazardly and generally did not appear to be aiming their shots. An example of the play of the best such network is shown in figure 9. In contrast, networks encoded with PEEL would move to position themselves on the other side of the ball from the goal and then either push the ball toward the goal or spin to kick it toward the goal. The best of these networks seldom missed shots and an example of its behavior is shown in the sequence of images in figure 10.

The NCOCB is also evident in the ANNs evolved for the goal-scoring task. The graph in figure 11 contains plots of the average connectivity of nodes from individuals in the final populations on the goal-scoring behavior. Once again the NCOCB can be seen by the strong bias toward higher connectivity on nodes created earlier in the graph construction process. The plots on this graph are not as smooth those from evolution on parity since in this case only

[3] Animations of goal-scorers controlled with ANNs evolved using both CEEL and PEEL are available online at: `http://ic.arc.nasa.gov/people/hornby/evo_control/evo_control.html`.



(a) (b)

(c) (d)

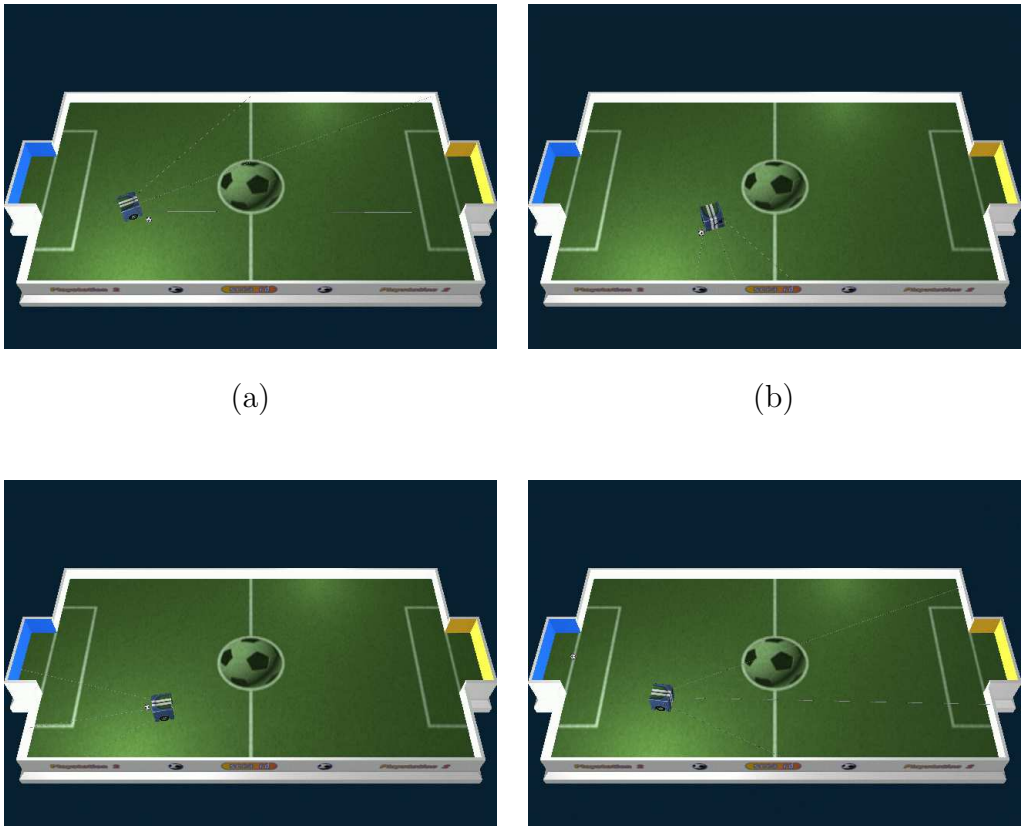
Fig. 9. A soccer player, evolved using the CEEL language, in action in the simulated soccer environment: (a) the soccer player is moving toward the ball, (b) while moving toward the ball it stops to spin in place (a frequent behavior), (c) it reaches the ball and starts to push it, (d) it ends up pushing it away from the desired goal.

Table 5
Average node connectivity and creation order at the end of evolution for evolved goal-scoring ANNs.

	In 1	In 2	In 3	In 4	In 5	In 6	In 7	Out 1	Out 2
	CEEL								
Connectivity	19.2	1.54	5.18	2.98	1.62	2.71	3.18	7.96	1.64
Order	2.72	7.60	12.1	14.6	19.0	26.1	27.2	15.4	25.4
	PEEL								
Connectivity	4.03	2.76	1.92	1.56	1.52	1.50	1.98	3.07	2.72
Order	1	2	3	4	5	6	7	8	9

four trials were run with each of CEEL and PEEL whereas the parity plots are the average of 500 trials each.

Examining the average connectivity of the output nodes of ANNs in the fi-



(a) (b)

(c) (d)

Fig. 10. A soccer player, evolved using the PEEL language, in action in the simulated soccer environment: (a) the soccer player is moving toward the ball, (b-c) it circles around the ball, and (d) it spins and shoots the ball into the goal then turns to the center of the field where the ball will reappear.

nal generation of evolution shows that there is a large difference in average connectivity between evolved ANNs encoded with CEEL and those encoded with PEEL (table 5). With ANNs encoded with CEEL there is a much greater number of connections to output 1 than there are to output 2, whereas with ANNs encoded with PEEL the average number of connections to the two output nodes is almost the same. This difference in connectivity may explain why CEEL-encoded soccer players frequently spin in place when it is not beneficial whereas PEEL-encoded soccer players drive around more smoothly.

The connectivity values listed in table 5 show that nodes created later in the construction process sometimes have a higher connectivity than nodes created before them. This can be understood by considering where in the tree-structured encoding the creation of these nodes occurred. Here the order of input nodes by connectivity is 1, 3, 7, 4, 6, 5, 2. Such a connectivity order could be obtained with input 1 at the top of the encoding-tree, input 2 by itself on one, small subtree and the rest of the input nodes on a second, much larger, subtree. In this second subtree input 3 would be at the root with input 7 on its

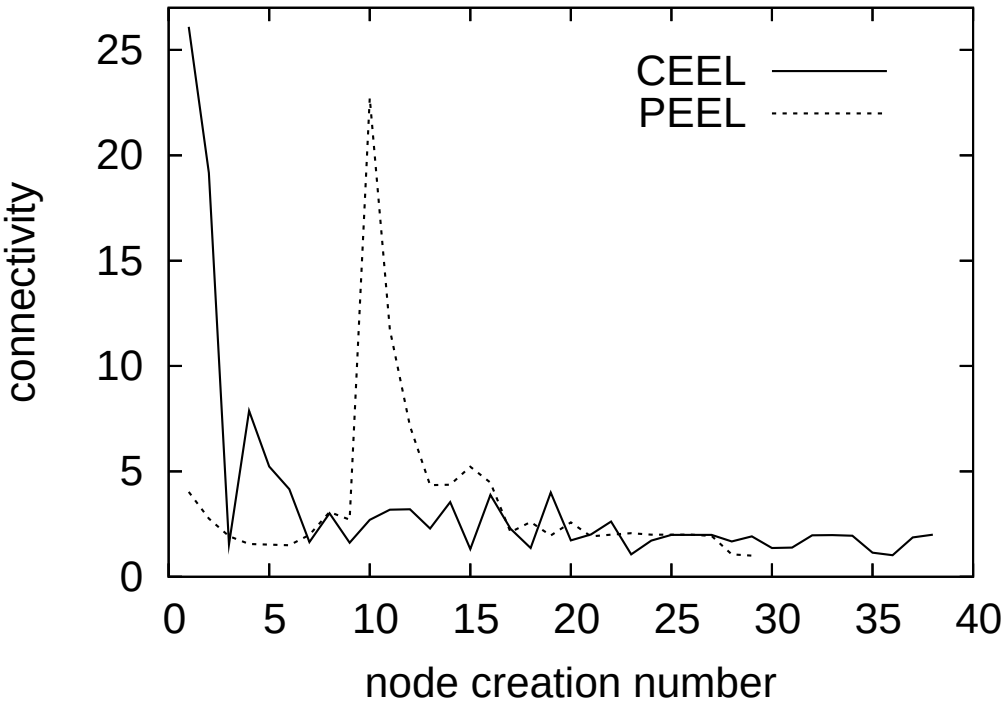


Fig. 11. Average NCOCB with CEEL and PEEL at the final stage of evolution on the goal-scoring behavior.

own small sub-subtree, and the rest of the inputs on a larger sub-subtree with input 4 at the top. Finally, on the sub-subtree with input 4 at the root, input 6 and input 5 would each be near the top of their own sub-sub-subtrees, with the one containing input 6 being larger than the one containing input 5. Thus connectivity is not strictly dependent on creation order rather it is determined by the size of the encoding-tree beneath the node-creating operator.

9 Discussion

In section 4 we showed that representing graphs with tree-structured EE languages produces a connectivity bias on nodes that is based on the order they are created in. Later we described two variations of EE languages for handling I/O nodes, one of which (CEEL) had the NCOCB and the other (PEEL) only had the NCOCB on hidden nodes. In comparing these two languages on calculating 3-parity and on a goal-scoring task we found that evolution with PEEL produced ANNs with higher fitness which suggests that the NCOCB has significant effect on evolutionary performance. Of interest is whether there is an EE language that does not have the NCOCB on both the I/O nodes and the hidden nodes.

While we described one implementation of an edge encoding language there are variations on what some operators do and also there are other operators that can be used. For example, the edge-encoding language of section 3 differs from Luke and Spector's [25] in that here edges are not explicitly deleted, rather they disappear if they are not assigned a weight, and the split operator used here does not delete the link $\overrightarrow{ab}$ when it creates the new neuron c and links $\overrightarrow{ac}$ and $\overrightarrow{cb}$. An operator for explicitly deleting links would not necessarily change the biases in resulting networks since the `no-op` operator with no children has the same effect. It is likely that that the split operator used here results in a stronger NCOCB than Luke and Spector's would since it adds links to existing neurons without removing any. Regardless, changes to either of these operators would not remove the NCOCB.

In fact, although different edge-operators will affect the degree to which the node creation order connectivity bias occurs, all edge-encoding languages with a tree-structured representation will have the NCOCB. At a given depth in the tree-structured genotype there is on the order of twice as many operators as at the depth immediately above. Graph nodes produced by operators that are leaf nodes in the genotype will have a small connectivity, which is typically one input and one output edge. Graph nodes produced by operators that are in the layer immediately above the leaf operators in the genotype will have their connectivity increased by one for each of their children operators that creates a node. By recursively going up the tree-structured genotype we see that the connectivity of nodes in the graph is based on depth in the genotype of the operator that creates them. Since there is a bias in the number of operators at a given depth in the genotype – one operator is at the root of the genotype and approximately half the operators are leafs – there is a node creation-order connectivity bias.

An exception to the above argument is if the edge-encoding language includes an operator for merging two nodes together. Adding an operator that merges two nodes into one and joins their inputs and outputs could reduce the bias somewhat because it allows for nodes created near the end of the construction process to be merged to create one with a connectivity higher than two. But, if instead of treating this as a newly created node it is treated as being the first node receiving the connections of another node then the addition of a merge operator would not remove the NCOCB, rather it would result in a stronger bias.

A solution to creating a representation for graphs which does not have the NCOCB comes from looking at the alternatives to tree-structured edge encodings. One option is to switch to operators in which the connectivity of a new node is not dependent on its depth in the genotype, but these would be node operators which have their own shortcomings [25]. Another option is to change to something other than a tree-structured genotype. While a linear genotype

will have the NCOCB to an even greater degree than a tree-structured genotype because strings are a special type of tree, a graph-structured genotype should not have the NCOCB. With a graph-structured genotype the challenge then would be to construct one that is both amenable to recombination and to which generative components can be added to improve scalability. An example of one such graph-structured, generative representation is VHDL [26], a hardware description language for describing circuits which has features of abstraction, hierarchy and modularity. A third option is to use a staged process in which first the nodes are created and then links are created. With this third option since all nodes exist before links are created there would not necessarily be a creation-order connectivity bias.

10 Conclusion

Various representations have been developed for encoding artificial neural networks, circuits and other graph structures and in our review we identified weaknesses with each them. Some were not generative, that is they did not have genotypic reuse through either iteration or abstraction, hence evolution with them would not scale beyond simple graph structures, or their iteration would repeat alleles in genes where they would have different meanings. Others were limited by such things as poor combinative ability, mutating a single element of the genotype would affect the entire phenotype disruptively. Finally, cellular encoding, one of the more popular methods, had been shown to have weaknesses in assigning weight values to edges and recombination of subtrees does not preserve the phenotypic sub-networks. This left edge encoding as a representation for encoding graph structures which had promise.

In this paper we analyzed edge encodings and identified shortcomings with this representation. First we showed that the connectivity of a node is strongly biased by the order of its creation, and we called this weakness the node creation order connectivity bias (NCOCB). The other weaknesses we identified are dependent on how input/output nodes are handled and showed two systems for connecting to I/O nodes, constructive edge encoding language (CEEL) and parameterized edge encoding language (PEEL). While CEEL can handle situations in which the number of I/O nodes is not fixed, PEEL is better able to create and maintain ANNs when the number of I/O nodes is fixed and it does not have the NCOCB with the I/O nodes.

To demonstrate the significance of the NCOCB on evolutionary performance we evolved ANNs for calculating 3-parity as well as for a goal-scoring task using both CEEL and PEEL. The results from these experiments showed that evolution of ANNs encoded using PEEL solved odd-3-parity more than twice as often as when using CEEL and evolution with PEEL produced goal-scorers

which had an average fitness of roughly twice that of those encoded using CEEL. These results suggest that the NCOCB is having a significant affect on evolutionary performance and a representation for ANNs which does not have the NCOCB at all would perform even better.

Since all tree structured edge encoding languages will have the NCOCB to some degree, of interest is directions to go for creating scalable representations without this shortcoming. Toward this end we identified two directions for further work, the first being to use graph-structured representations and the second being a two-phased approach to graph construction. Whichever direction future work takes in designing new representations for encoding ANNs, the designers will need to be aware of the different shortcomings that have been identified for existing ones.

Acknowledgements

Some of this research was conducted while the author was at Sony Computer Entertainment America, Research and Development as well as Brandeis University. The soccer game and simulator was developed by Eric Larsen at SCEA R&D.

References

[1] P. J. Angeline. Morphogenic evolutionary computations: Introduction, issues and examples. In J. McDonnell, B. Reynolds, and D. Fogel, editors, *Proc. of the Fourth Annual Conf. on Evolutionary Programming*, pages 387–401. MIT Press, 1995.

[2] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[3] R. D. Beer and J. G. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992.

[4] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies of an evolutionary design problem. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Genetic and Evolutionary Computation Conference*, pages 35–43. Morgan Kaufmann, 1999.

[5] E. J. W. Boers, H. Kuiper, B. L. M. Happel, and I.G. Sprinkhuizen-Kuyper. Designing modular artificial neural networks. In H.A. Wijshoff, editor, *Proc. of Computing Science in The Netherlands*, pages 87–96, SION, Stichting Mathematisch Centrum, 1993.

[6] K. E. Drexler. Biological and nanomechanical systems. In C.G. Langton, editor, *Artificial Life*, pages 501–519. Addison Wesley, 1989.

[7] F. D. Francone, M. Conrads, W. Banzhaf, and P. Nordin. Homologous crossover in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1021–1026, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[8] K. Fredriksson. Genetic algorithms and generative encoding of neural networks for some benchmark classification problems. In *Proc. of the Third Nordic Workshop on Genetic Algorithms and their Applications*, pages 123–134. Finnish Artificial Intelligence Society, 1997.

[9] C. M. Friedrich and C. Moraga. An evolutionary method to find good building-blocks for architectures of artificial neural networks. In *Proc. of the Sixth Intl. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 951–956, Granada, Spain, 1996.

[10] F. Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Ecole Normale Supérieure de Lyon, 1994.

[11] F. Gruau. Automatic definition of modular neural networks. *Adaptive Behavior*, 3(2):151–183, 1995.

[12] P. C. Haddow, G. Tufte, and P. van Remortel. Shrinking the genotype: L-systems for EHW? In Y. Liu, K. Tanaka, M. Iwata, T. Higuchi, and M. Yasunaga, editors, *Proc. of the Fourth International Conference on Evolvable Systems: From Biology to Hardware*, Lecture Notes in Computer Science; Vol. 2210, pages 128–139. Springer-Verlag, 2001.

[13] G. S. Hornby. *Generative Representations for Evolutionary Design Automation.* PhD thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA, 2003.

[14] G. S. Hornby. Shortcomings with tree-structured edge encodings for neural networks. In K. Deb et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, Vol. II*, LNCS 3103, pages 495–506, New York, 2004. Springer-Verlag.

[15] G. S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In H.-G. Beyer et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, GECCO-2005*, pages 1729–1736, New York, 2005. ACM Press.

[16] G. S. Hornby and B. Mirtich. Diffuse versus true coevolution in a physics-based world. In W. Banzhaf and et al., editors, *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1305–1312. Morgan Kaufmann, 1999.

[17] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002.

[18] G. S. Hornby, S. Takamura, O. Hanagata, M. Fujita, and J. Pollack. Evolution of controllers from a high-level simulator to a high dof robot. In J. Miller, editor, *Evolvable Systems: from biology to hardware; Proc. of the Third Intl. Conf.*, Lecture Notes in Computer Science; Vol. 1801, pages 80–89. Springer, 2000.

[19] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[20] J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.

[21] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Automated WYWIWYG design of both the topology and component values of analog electrical circuits using genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, Stanford University, CA, USA, 1996. MIT Press.

[22] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[23] A. Lindenmayer. Mathematical models for cellular interaction in development. parts I and II. *Journal of Theoretical Biology*, 18:280–299 and 300–315, 1968.

[24] J. Lohn, D. Gwaltney, G. Hornby, R. S. Zebulum, D. Keymeulen, and A. Stoica, editors. *The 2005 NASA/DoD Conference on Evolvable Hardware*, Washington, DC, 2005. IEEE Press.

[25] S. Luke and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. In J. Koza, editor, *Late-breaking Papers of Genetic Programming 96*, pages 117–124. Stanford Bookstore, 1996.

[26] J. P. Mermet, editor. *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, Order Code SH 16840*. The Institute of Electrical and Electronics Engineers, New York, NY, 1994.

[27] J. F. Miller and P. Thomson. A developmental method for growing graphs and circuits. In A. M. Tyrrell, P. Haddow, and J. Torrensen, editors, *Fifth Intl. Conf. on Evolvable Systems: From Biology to Hardware*, Lecture Notes in Computer Science; Vol. 2606, pages 93–104. Springer, 2003.

[28] S. Nolfi and D. Floreano, editors. *Evolutionary Robotics*. MIT Press, Cambridge, MA, 2000.

[29] S. Nolfi and D. Parisi. Growing neural networks. Technical Report PCIA-91-15, Institute of Psychology, CNR Rome, 1992.

[30] S. Nolfi and D. Parisi. Evolving artificial neural networks that develop in time. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, editors, *European Conference on Artificial Life*, pages 353–367. Springer, 1995.

[31] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In E. Goodman, editor, *Proc. of the Seventh Intl. Conf. on Genetic Algorithms*, pages 18–25, San Mateo, California, 1997. Morgan Kaufmann.

[32] R. Poli and W. B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[33] A. Siddiqi and S. Lucas. A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *IEEE International Conference on Evolutionary Computation*, pages 392–397, Piscataway, NJ, 1998. IEEE Press.

[34] A. Witkin and D. Baraff. Physically based modeling: Principles and practice. Online Siggraph '97 Course notes, http://www-2.cs.cmu.edu/∼baraff/sigcourse/index.html.

[35] R. S. Zebulum, D. Gwaltney, G. Hornby, D. Keymeulen, J. Lohn, and A. Stoica, editors. *The 2004 NASA/DoD Conference on Evolvable Hardware*, Seattle, WA, 2004. IEEE Press.